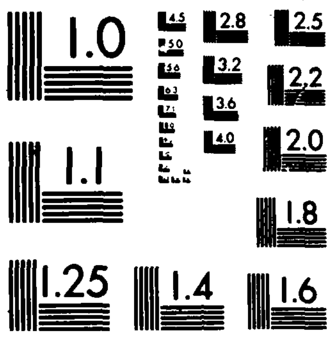

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(1)

Working Paper No. 347

DTIC FILE COPY

Comparative Analysis of Mathematical Programming Systems

by

Sergio V. Maturana

AD-A182 485

May 1987

WESTERN MANAGEMENT SCIENCE INSTITUTE
University of California, Los Angeles

87 7 13 125

WESTERN MANAGEMENT SCIENCE INSTITUTE
University of California, Los Angeles

Working Paper No. 347
May 1987

Comparative Analysis of Mathematical Programming Systems

by

Sergio V. Maturana

Abstract

There is a growing number of mathematical programming systems that try to offer a simpler way for solving complex problems, using the computer, than the traditional approach. This paper undertakes a comparative analysis of some of these systems and identifies the main issues involved in designing and implementing such systems. [illegible] ; computer programs ; computer architecture ; user interface ; systems [illegible]

Presented at the TIMS/ORSA Meeting in New Orleans, LA, May 1987.
I acknowledge with gratitude the constructive comments of Arthur Geoffrion, Linda Leon and John Mamer. This work was partially supported by the National Science Foundation and the Office of Naval Research. The views herein are mine and not to be attributed to the sponsoring agencies.

Contents

A-1

1 Introduction

It is a well known fact that, despite the existence of very good mathematical programming solvers, relatively few people use them. One of the main reasons for this lack of use is that the potential user must spend a large amount of time learning to use the solver. Furthermore, most of the solvers are capable of solving only one type of problem and in general, there are few similarities among different solvers. To make things even worse, the process of building the input file for most solvers is complicated and error-prone (e.g., typically a matrix generation program is needed).

However, technological advances have shifted the economics of software development. To quote Drud [8]:

> "The decreasing price of computing power relative to manpower together with the speed with which models change has made the traditional approach of representing models by special purpose computer programs inefficient."

Technological advance has been one of the main forces behind the growing effort over the last few years to better support the whole modeling process of which finding a numerical solution is just one part. As a result of this effort, the phrase "integrated modeling system" has become quite popular. In [14], Geoffrion gives the following interpretation of this phrase:

> "The phrase has two parts. I take 'integrated' to mean *uniting things that can stand alone usefully but that are even more useful when put together;* and 'modeling system' to mean *computer software that supports (part of) the modeling life-cycle.*"

The scope of this survey will be restricted to mathematical programming oriented integrated modeling systems. These systems will be referred to as "mathematical programming systems" in the remainder of this paper.

This paper undertakes a comparative analysis of some of the mathematical programming systems that have tried to simplify the use of mathematical programming solvers. The purpose of this survey is to identify the main issues involved in designing and implementing such a system. It is worth noting that some of the systems surveyed have attempted to support more parts of the modeling life-cycle than others. However, the emphasis in all of these systems is towards supporting the solution part of the modeling life-cycle.

In [13] Geoffrion discusses some modeling systems related to *Structured Modeling*[1] among which there are several mathematical programming systems. The systems are discussed in the context of the eight desirable modeling system features proposed in [13], Section 1.2. They are repeated here for the reader's convenience.

a) a rigorous and coherent conceptual framework for modeling based on a single model representation format suitable for managerial communication, mathematical use, and direct computer execution

b) independence of model representation and model solution, with model interface standards to facilitate building a library of models and of easily accessed solvers for retrieval, systems of simultaneous equations, optimization, and other important manipulations

c) sufficient generality to encompass most of the great modeling paradigms that MS/OR and kindred model-based fields have developed for organizing the complexity of reality (activity analysis, decision trees, flow networks, graphs, markov chains, queueing systems, etc.)

d) usefulness for most phases of the entire life-cycle associated with model-based work

e) representational independence of general model structure and the detailed data needed to describe specific model instances

f) desktop implementation with a modern user interface (e.g., visually interactive, directly manipulative, syntactically humane, and with liberal use of graphics and tables)

g) integrated facilities for data management and ad hoc query in the tradition of data base systems

[1]For an introduction to *Structured Modeling* see [12].

h) immediate expression evaluation in the tradition of desktop spreadsheet software.

The present survey is in certain ways similar to the one undertaken by Geoffrion in [13]. However, there are some important differences. The present survey is primarily concerned with mathematical programming systems, while Geoffrion also deals with other types of systems. Also, the present survey is oriented more towards the implementational aspects of the systems.

The mathematical programming systems that will be surveyed, using the characteristics just described, are the following:

- *GAMS*
- *GINO*
- *GXMP*
- *HEQS*
- *IFPS/OPTIMUM.*

This is by no means an exhaustive list of what can be considered to be mathematical programming systems, but it is representative of what currently exists. Other mathematical programming systems that could have been included are: *AMPL* [11], *CAMPS* [20], *EMP* [24], *LINDO* [25], *LPMODEL* [16], *MLD* [4], *PAM*, *PLATOFORM* [21], *UIMPS* [9] and *What's Best!*. The objective of this paper is not to do a comprehensive survey of existing mathematical programming systems, but rather to have enough diversity in the systems surveyed so that all the important implementational issues can be identified and discussed.

Our analysis of the mathematical programming systems listed above should identify and shed light upon the most important issues that have to be resolved to design and implement a *"good"* mathematical programming system. This should set the stage for the author's research on optimization interfaces for *Structured Modeling* systems.

2 Analysis Criteria

To organize our survey of mathematical programming systems, we will use the following eight characteristics, most of which are related to the eight desirable features cited in [13]. We chose not to use the cited features, since the bias of this survey is towards implementational problems while the desirable features cited in [13] are more user oriented.

2.1 Mathematical Programming Solvers Used

In [14], Geoffrion defines a solver as a "...method or program for purposefully manipulating a model." He also identified the following types of manipulation:

- Definitional Calculation (as in a spreadsheet)
- Satisfaction (solve a system of linear or nonlinear equations and/or inequalities)
- Optimization
- Query Processing (as in a data base management system)
- Inference (as in artificial intelligence)

Most of the mathematical modeling systems surveyed supported mainly the third type of solver, namely Optimization. However some of the systems also supported, in some degree the first two types of solvers. Also, some systems provide some query processing capabilities. However, these capabilities will be dealt with in section 2.5. This section will deal with the Satisfaction and Optimization type of solvers while section 2.7 will analyze the Definitional Calculation capabilities of each system.

Traditionally, the solver has been the most important part of a mathematical programming system. However, as noted earlier and despite the existence of very good optimization solvers, the number of users of such systems is not very large. This suggests that other factors, like the user interface, the ability to use a modeling language, etc., are also critical in order to make the system easy to use by a wide audience. Nevertheless, the optimization solver is still the most important part of a system; even though many users may not be aware of the sophisticated solver they are using, they judge the power of the solver by its results.

One of the problems cited earlier for the lack of use of mathematical programming solvers is that they tend to be rather specialized. That is, in many instances a solver that works well in some kinds of problems will work poorly, or not all in other kinds of problems. This raises the problem of solver selection, which is related to *feature (b)* — independence of model representation and model solution. Therefore, having the ability to easily build a library of solvers that can be accessed by the system allows the system or user to select the appropriate solver for the problem. Also, as the solver technology improves, we will be able to add better solvers to the library. Note that this is also related to the system architecture, because if the solver and the rest of the system are not well separated, updating the solver or adding a new one becomes much more difficult. In fact, Drud [8] recommends deciding at an early stage in the design phase of the modeling system, if *outsiders* will be allowed to add extra algorithms to the system.

In order to have an open library of solvers, it is necessary to have a "standard" interface between the modeling system and the solver. This standard representation must be carefully designed, since if it has to be changed at a later time, all the solvers in the solver library would have to be modified. A standard interface is much harder to achieve for nonlinear programming solvers, since there is no de-facto standard for input files as there is for linear programming codes — namely the MPS format[2].

2.2 System Architecture

The system architecture is how the different parts of a computer program interact. For example, a computer program can be a monolithic program, with no distinguishable parts, or it can be a collection of separate programs — or modules — that interact via system files and which can be coordinated through a higher level program (e.g., *Scripts* in *UNIX* environments).

In general, the user doesn't have any idea about the system architecture of the system he is using — and probably doesn't care as long as the system works. From the implementor's point of view, however, the system architecture design is one of the first and most important tasks of the development process. The system architecture design will strongly influence the effort and time required

[2]Even though the MPS format is the de-facto standard for specifying linear programming input files, it has many strong critics. However, from the implementor's point of view, a bad standard interface may be better than no standard at all.

to complete each of the stages in the system development.

One of the most recommended approaches for designing a system [19] is the modular design. The basic idea of the modular approach is to break down a large program into smaller ones. Well designed modules should be loosely connected or coupled to other modules. The concepts of top-down development or step-wise refinement and structured programming are closely related to modular design.

Using a modular approach has the following advantages:

- Easier development. The task can be divided into more manageable parts which can be developed independently.

- Greater flexibility. Not only are the features mentioned before hard to implement, but also different users may want these features implemented in different ways. This means that the system should also be quite flexible. A well designed modular approach could make the system much easier to adapt to changing needs and different types of users.

- Easier debugging. In a well designed modular system, the modules should be as self-contained as possible and interact mainly through standard files as much as possible. This makes it possible to test and debug each module independently.

- Easier implementation of an open solver library. A modular system can be implemented so that adding a new solver can be achieved by adding a module that translates the internal representation of the problem into the representation used by the new solver. The solver itself can be another module.

A good example of a system developed using a modular approach is *HEQS*, which by no coincidence runs under *UNIX*. This operating system — which has become increasingly popular — makes developing loosely coupled, self-contained programs relatively easy. Most other operating systems do not support this approach very well.

A consequence of having a modular architecture is that it becomes necessary to have multiple representations of the model. For example, there must exist a user representation. (All of the systems surveyed used a text representation for entering the model.) This user representation has to be easy to read and understandable by humans. One of the first tasks of a modeling system is to convert this user representation to an intermediate representation of the model. This representation should be such that the rest of the system can manipulate the model in a much more direct way than with the text representation. Another necessary representation of the model is the input file required by the solver to find a solution to the model. Having multiple representations means there have to exist translator programs that can translate the user text representation into the intermediate representation, and the intermediate representation into a solver representation and back. Each of these translator programs can be implemented as a separate module.

Another approach is building upon an existing system as opposed to designing and implementing a stand-alone system from scratch. If the system is designed from scratch, it is possible to achieve a much better integration of the different functions of the system. On the other hand, building upon an existing system has the advantage of not having to design and implement complex systems that have already implemented important parts of the complete system such as the user-interface and data management modules. This approach greatly depends upon the selection of the base system. Examples of systems which have been built upon other systems are *IFPS/OPTIMUM*, built upon *IFPS*, and *What's Best!*, built upon *LOTUS 1-2-3*.

Unfortunately for this survey, not much information is available about the system architecture of most of the systems surveyed.

2.3 Modeling Language Features

A great part of the ease of use and usefulness of the mathematical programming system will depend on the features of the Modeling Language. Designing a good modeling language that has *feature (c)* is probably the most difficult part of the implementation of a mathematical programming system.

Special attention will be given in this survey to the index referencing capabilities which tend to be weak in many modeling systems and which are critical in order to achieve sufficient generality.

In order to be usable by many people, the Modeling Language should be powerful enough to represent, easily and naturally, the problem under study;

and it should be easy to learn. Unfortunately, these two features tend to be contradictory.

Also, as noted by Dolk [7]:

> "The major trade-off in designing a modeling language for MP problems is the power of the language versus the difficulty in transforming the equations specified in the language to the appropriate representation for solution."

Most of the power, and problems of modeling languages, seem to refer to the indexing of variables. It is hard to have the generality of referring to the whole set of indices at some times, and at others to only a restricted subset. To quote Brooke et al [3]:

> "...we are probably dealing with multidimensional quantities such as demand in different regions for different products, production of different products in different plants in different periods, etc. Basic data, derived coefficients, variables and equations can all be multidimensional. It is also important to realize that exceptions exist in all but the most trivial models: initial and terminal periods differ, some plants may only produce some products, and so forth."

On the other hand, the structuredness of the modeling language has a strong bearing on how easy it is to learn and use. Non-procedural languages, which require very little structure, allow a great degree of freedom and are generally easier to learn. Procedural languages require much more structure and are harder or learn but they enable the system to detect more possible model formulation errors than non-procedural languages.

The type of functions allowed in the language is also an important point. Some systems have a large number of functions available while other systems supply only relatively basic functions.

The ability of defining Macros seems a desirable feature that has not been very exploited. *HEQS* has a nice implementation of this feature.

2.4 Model Management Features

Since the mathematical programming system probably will be used to solve many models and the same model can be used by many persons, it is important to be able to store, retrieve and manipulate models in a multi-model, multi-user context. As stated in *feature (d)*, the system should be useful for most phases of the entire life-cycle of the model.

One of the most important features associated with model management is the ability of a system to handle models in the same way that data base management systems (DBMS) handle data. Again quoting Dolk [7]:

> "One of the key assumptions underlying model management is that models, like data, are an important resource of an organization and should be managed with as much rigor and attention as data."

If models are considered to be like data, a Model Management system should have the same functional capabilities of a Data Base Management System (DBMS):

1. Model Description,
2. Model Manipulation,
3. Model Control.

Model description is the process by which the user specifies the model. A model management system supports different types of interfaces for this process, depending on the type of user, e.g. model builder versus decision maker.

Model manipulation includes the standard functions CREATE, STORE, DELETE, MODIFY and DISPLAY, as in data base management systems, but also adds model-specific operations such as EVALUATE, SOLVE, LINK, AGGREGATE, DECOMPOSE and WHAT-IF.

Model control involves issues of access authorization, security and privacy, integrity and administration. Even though these functions also exist in data base management systems, they can be much harder to implement in model management systems.

2.5 Data Management Features

Mathematical programming models typically require large amounts of data as input. In most instances, the data have to be manipulated in order to generate the necessary input for the model. This suggests the need of having built-in data management capabilities in the system, as stated in *feature (g)*. In order to take good advantage of this capability, however, it is important that the system supports independence between data and model representation — *feature (e)*.

A good modeling practice is to allow the user to enter the data needed to solve the model in its most basic form and then define the data transformations required to create the coefficients of the model. This should be supported by the data management capabilities of the system. It is generally agreed that handling the data in order to generate these coefficients is the task of the modeling process that takes the lion's share of time and effort.

There is a trend, among modeling systems, towards using the relational approach to data base management. This has the advantage of having considerable generality and of being easy to visualize. However, if the organization handles its data using IMS or CODASYL architecture, there may be problems involved in importing and exporting the data between the modeling system and the organization's DBMS. Also, since the relational model requires *flat* data structures, a hierarchical data base will require significant redundancy. In fact, any data base can be transformed into a relational or flat data base by introducing additional redundancy[5].

An alternative to having a built-in data base management system is to have the ability of using an external DBMS. If this could be achieved in such a way that it is transparent to the user, it would be as useful as having a built-in DBMS. However, this transparency is difficult to achieve, especially in a personal computer operating system.

2.6 User Interface

With the advent of Personal Computers, the standard of what is considered to be a good user interface has risen dramatically. Also, many users expect to be able to run any program on their desktop computers. Therefore, a good mathematical programming system has to attempt to meet the requirements specified in *feature (f)*.

Until recently, as noted by Gill et al [15], "... most optimization software has been developed in the form of FORTRAN subroutines which must be incorporated into the user's own programs." This is not acceptable for most PC users. Especially considering that FORTRAN compilers for personal computers tend to be not as good as those available for mainframes.

2.7 Intermediate Expression Handling

As mentioned in section 2.1, most of the mathematical programming systems support mostly the optimization type of solver, although it is possible to perform satisfaction and evaluation (definitional calculation) using an optimizer. Satisfaction can be achieved simply by leaving out the objective function and evaluation can be carried out by leaving out the objective function and specifying a rather trivial set of equations as constraints. However, if the system can recognize these types of models, it can carry out the solution process more efficiently.

More importantly, there may be a significant difference in how models are represented by the modeling language and the modeling paradigm used by an optimization solver. Almost all optimization solvers assume the existence of *decision variables*, *constraints* and an *objective function*. The constraints are equations or inequalities that involve the decision variables and the objective function is a function that depends on one or more decision variables. The solver then tries to find which values of the decision variable maximize or minimize the objective function while still satisfying the constraints. On the other hand, many modeling languages encourage the use of variables other than the decision variables, as is stressed in [23], "...the presence of many intermediate and exogenous variables is an important feature of optimization models specified by planning languages." These variables, some times called parameters or exogenous variables, offer two important advantages: they allow representing models with a higher independence of the detailed data and they encourage entering data in its most basic form with the transformations defined within the model.

The presence of exogenous variables can result in *intermediate expressions* which are expressions which involve only numeric values or other exogenous variables. The importance of these expressions is that they can be *evaluated* without solver intervention. Also, in some cases a more efficient solver may be used. For example, a model may have nonlinear expressions that suggest the need for a nonlinear solver, yet if these nonlinear expressions are recognized as intermediate expressions and are evaluated, the resulting problem could be linear and therefore a much more efficient solver could be used. In general, almost any optimization solver will be much faster by recognizing and eliminating intermediate expressions. However, nonlinear programming solvers that have to perform many evaluations tend to benefit much more than linear programming solvers. On the other hand, as noted before, if intermediate expressions are not recognized, they could introduce nonlinearities in an otherwise linear problem, therefore forcing the use of a nonlinear solver when a linear one could have been used if the intermediate expressions would have been eliminated.

The handling of intermediate expressions involves three parts: classifying variables into decision and exogenous variables, identification of intermediate expressions and evaluation of these expressions. There are some important differences in how this can be done. For example, a system can take a rather procedural approach and require the explicit definition or declaration of which variables are decision variables, while other systems may assume that any variable which is not an exogenous variable is a decision variable. Also, some systems, like *IFPS* and *HEQS* may allow circular references among exogenous variables. Thus, these systems have to have the ability to solve sets of linear and nonlinear equations in order to eliminate all intermediate expressions. Spreadsheet technology does not in general support circular references in this way, even though it is possible to solve systems of equations using spreadsheets.

In order to better understand how intermediate expressions are handled, it is useful to see how modeling systems handle symbolic expressions in general. First of all, most modeling systems represent symbolic expressions in three parts [8].

The first part gives the length of the next two parts, the second part describes the arguments of the expression and the third part describes the operations. The second part can be thought of as a table containing at least three columns. The first column is the argument index which is simply the internal name of the argument. In most cases this is simply an integer number. The second column gives the type of the argument, e.g. variable, numeric constant, etc. Finally, the third column has the name of the argument, if it is a variable, or the numeric value if it is a numeric constant.

The third part of the symbolic expression description can be thought of as a table with at least four columns. The first column refers to the first operand, the second column refers to the second operand, the third column refers to the operation and the fourth column is the result column. The operations are generally identified by a numeric code. The operands on the other hand, can either refer to the internal name of an argument — the first column of the argument description table — or to the result internal name.

This representation is constructed while parsing the model representation while also checking the modeling language syntax. Once the model representation has been parsed, the system analyzes the dependencies among the variables. It is at this stage when circular references can be detected. Also at this stage, the system may classify the variables as endogenous or exogenous. Finally, if the system has evaluation capabilities, if will find an order in which to carry out the evaluations. If the system can handle systems of equations, it may identify the independent subsets of equations and the order in which they have to be solved.

The way in which modeling systems and spreadsheets handle symbolic expressions is very similar. In fact, a spreadsheet can be thought of as a two dimensional array of variables which can reference each other. This similarity has been exploited by systems like *IFPS/OPTIMUM*, *What's Best* and *VINO* that use spreadsheet, or spreadsheet related technology, as a *front-end* for an optimizer. Since spreadsheets store expressions as described above, it is possible to efficiently translate a spreadsheet model formulation into the optimizer input file. However, as noted before, most visual spreadsheets (*LOTUS 1-2-3*, *Visicalc*, etc.) are unable to handle circular references but are much more interactive than modeling systems.

2.8 User Control

An important characteristic of a mathematical programming system is the degree of user-control it requires and how much user-control it allows. All systems require some user-control and at the same time, all systems limit the options a user can exercise when using the system.

The mathematical programming systems surveyed here are aimed at a wide range of users with very different backgrounds. A good mathematical programming system should be able to execute the optimization process as automatically as possible, to support the user for whom the solver is a *"black box"*.

At the same time, it should allow a high degree of control by the technically knowledgeable user.

Most optimization programs require the specification of problem-dependent parameters. To quote Gill et al [15]:

> "In an ideal world, these (problem-dependent parameters) would be chosen by the user, who should be in the best position to understand the problem. However, many users do not wish to expend the effort to analyze the innermost details of their problem (or do not know how)."

In particular, for those systems that have more than one solver, the problem of which solver to use can be dealt with by either the user or the system. In this context, the problem of linearity determination is very important since linear programming solvers are much faster than non-linear programming solvers. Therefore, if the system has to determine which solver to use, it has to have the ability to determine whether the problem is linear or not. Similarly, the system could be able to detect special structures, like network types of problems, in order to be able to use more efficient solvers for those problems.

There is an obvious trade-off between having automatic execution of the system and having more user-control. If the user is very naive, or the model is relatively easy to solve, having automatic execution makes the system easier to use. On the other hand, if the user is knowledgeable, and the model is hard to solve, having a greater degree of user-control could lead to a big improvement in efficiency.

The advantages of having automatic execution of the solver process are quite obvious. However, some people note that this approach has some important drawbacks: The unsophisticated user could easily develop a model which would be very hard to solve when maybe a much simpler model could have been appropriate. The situation is somewhat similar to when the first compilers were introduced. Many believed that a computer program would never be able to produce code efficient enough to displace professional programmers producing machine code by hand. Today, very few applications are completely programmed in machine language.

A good balance between the two seems to be having a control-table in which the user can over-ride the system. If the user doesn't enter anything in the control-table, the system assumes default values. Gill et al [15], offer a slightly different alternative:

> "We hope that future software will provide a user interface that is as simple as possible for the majority of users, yet retains some flexibility for the knowledgeable user to benefit from knowledge of the problem. With FORTRAN subroutines libraries, this has been achieved by providing two levels of software. The user who wants simplicity can invoke an outer, 'easy-to-use' version, which sets default values and calls automatic procedures to choose problem-dependent quantities; the user who wishes to 'tune' an algorithm to a given problem has access to an inner routine in which these quantities can be specified as parameters."

3 Systems Analyzed

In this section, each of the systems in our survey is analyzed according to the criteria stated in section 2.

3.1 GAMS

GAMS — for General Algebraic Modeling System — has been under development at the World Bank for a number of years for use primarily in optimization-based economic development studies [2,17].

This program has been implemented for several mainframe computers and also has a version for the IBM PC [22].

Mathematical Programming Solvers Used

The mainframe version of *GAMS* has been interfaced with 4 different linear programming systems. It also has been interfaced with the following nonlinear programming systems [3]:

- *GRG2*
- *CONOPT*
- *NPSOL*

- *MINOS 5.0*

The IBM PC version of *GAMS* interfaces only with the *MINOS* optimization package — MINOS 3.4 for linear programming problems and MINOS 5.0 for nonlinear problems.

Despite the relatively large number of solvers available, there is no standard interface between the *GAMS* system and a library of mathematical programming solvers. Each interface has to be tailored to the specific solver.

System Architecture

Not much has been published about the *GAMS* system architecture. However, in [26] the "*GAMS* Machine" is described:

> "A *GAMS* program is compiled by the *GAMS* compiler, which produces a set of instructions for carrying out the intent of the source program. The instructions produced by the *GAMS* compiler are not tied to any single computer hardware .... Rather, the *GAMS* instructions are for execution on a non-existent computer which we call the *GAMS* machine."

In [26] the following *GAMS* data structures for the Execution system are also described:

1. The Symbol Table
2. The Unique Elements Table
3. The Instruction Format
4. The Program Counter
5. The Execution System Stacks
6. Miscellaneous Variables

The Symbol Table is created by the compiler and contains every symbol (name of set, parameter, variable, equation, etc.) mentioned in a *GAMS* source program. The Symbol Table can be accessed quickly using an index number, if this is known. If the index number is unknown, there is a hash function which manipulates the symbol name to return an index into a hash table. Each hash table entry points to a linked set of hash records. Each hash record contains a symbol index and a pointer to the next hash record.

The Unique Elements Table is created by the compiler and contains every unique alphabetic entry (set element, parameter element, etc.) mentioned in a *GAMS* source program.

The Instruction Format is an array of elements of fixed length. Each element corresponds to an instruction.

The Program Counter is an index which is used to indicate the instruction being executed in the array code.

The Execution System operates with three stacks: the control stack, the index stack and the value stack. The control stack is used for controlling loops, the index stack contains unique element numbers and the value stack contains the real values for unary and binary operators as well as the fetch and store operations.

The Miscellaneous Variables are used for diagnostics and options.

Finally, [26] gives the definition of the *GAMS* Machine Instruction Set.

Modeling Language Features

GAMS' modeling language follows closely the Mathematical notation. To quote Bisschop and Meeraus [2]:

> "... a model builder must be able to express all the relevant structural and partitioning information contained in a model in a convenient short-hand notation. We strongly believe that one can only accomplish this by adhering to the rigorous and scientific notation of algebra. ... humans with a basic knowledge of algebra can use it as the complete documentation of their model. In addition to this, the algebraic notation contains all the necessary information that is needed for an automatic interface with the various linear and nonlinear solution routines."

The basic building blocks of *GAMS*' modeling language are:

- SETS

- PARAMETERS
- VARIABLES
- EQUATIONS

A simple (one-dimensional) set in *GAMS* is a finite collection of labels. These sets play an important role in the indexing of algebraic statements.

The following important set operations can be performed in *GAMS*' modeling language:

- Union
- Intersection
- Difference

These operations may be done with or without the existence of a superset. However, if a superset exists, the operations can be expressed much more succinctly.

GAMS also has two very useful set operators: `CARD` and `ORD`. The `CARD` operator returns the number of elements in a set and `ORD` gives the ordinal position of an element in a set.

A very important operator, in the *GAMS* language, is the "$" or "*such that*" operator. The definition of this operator is the following: let A be a name or an expression in *GAMS*, and let B be a name or true-false expression. Then the phrase A$B is a conditional statement in *GAMS* where the name A is considered or the expression A is evaluated if and only if the name B is defined or the expression B is true. The dollar operator is very versatile and can be used to handle different types of complexities that arise in models.

The *GAMS* language has many built-in functions:

- ABS(X) Absolute value of X.
- CEIL(X) the integer just larger than X
- EXP(X) e raised to the power of X
- FLOOR(X) the integer just smaller than X

- LOG(X) the natural logarithm of X
- LOG10(X) the logarithm to base 10 of X
- MAPVAL(X) to map special values such as NA and INF
- MAX(X1,X2,...,XN) the largest of the elements X1,X2,...,XN
- MIN(X1,X2,...,XN) the smallest of the elements X1,X2,...,XN
- MOD(X,Y) remainder of `X/Y = X - FLOOR(X/Y) * Y`
- NORMAL(X,Y) returns a random number drawn from a normal distribution with mean X and variance Y
- POWER(X,Y) X raised to the power of Y, where Y has to evaluate to an integer
- ROUND(X,Y) X rounded off to the digit Y
- SIGN(X) the sign of X, i.e. 1 if X is positive, −1 if X is negative and 0 if X is zero
- SQR(X) X raised to the power of 2
- SQRT(X) the square root of X
- TRUNC(X) the truncated value of X
- UNIFORM(X,Y) returns a random number drawn from a uniform distribution with range X to Y

Model Management Features

GAMS does not have many model management features. However *GAMS* does encourage self-documentation by allowing comments inside the general model structure.

Also, a library of models can be kept, but this is only of limited utility since *GAMS*' models tend to be problem-specific because of the lack of separation between detailed data and model structure.

Data Management Features

In a *GAMS* model, the data — which are specified as Sets and Parameters — are **NOT** separated from the general model structure. They must be included along with the variables and equations of the model.

GAMS' modeling language has the following data types:

- SCALAR
- LIST
- TABLE

A SCALAR in *GAMS* is simply a number. A LIST is a one-dimensional array, while a TABLE is a two-dimensional array. Three and higher dimensional arrays can also be handled, but they are not as easy to represent in the model, since *GAMS* favors *"flat"* data structures.

As noted by Kendrick and Meeraus in Chapter 19 of [17], *GAMS* has relational data base capabilities:

> "*GAMS* stores data as a relational data base which can be used as a stand-alone data base system. The data are stored in relationships which are defined over domains. Queries can be answered from this data by using projection and join operations and relationships."

Even though as noted above, *GAMS* could be used as a stand alone data base system, in practical terms that would not be a good idea, because any change in the data base would mean having to recompile the data base. This would not be acceptable for any relatively large data base. Also, *GAMS*' "query language" is not very user friendly.

GAMS has extensive data transformation capabilities and it strongly encourages entering the data in its most basic form and defining all the needed transformations using *GAMS*.

User Interface

GAMS has a relatively plain user interface. It is command driven and mainly batch oriented. However it does have very good error recovery and reporting capabilities.

Intermediate Expression Handling

GAMS supports entering each piece of information only once and in its most basic form. The user can specify data transformations on the elemental data and perform some data validation.

GAMS has the following variable types:

- SETS
- PARAMETERS
- VARIABLES
- EQUATIONS
- MODELS

Each identifier in the model has to be one of these data types and has to be declared in the model representation. Elemental data is of type PARAMETER, and can be input using the PARAMETER declaration. Alternatively, the keywords SCALAR and TABLE can be used, since their default type is PARAMETER. Any variable that will be used to define data transformations has to be declared as PARAMETER type. The transformation associated to these variables is specified by using the assignment operator ("="). The expression to the right of this operator can contain numeric values and references to other variables of type PARAMETER.

Decision variables, on the other hand, are declared explicitly using the VARIABLE keyword and can have modifiers that specify the range of the variable, e.g. FREE, POSITIVE, NEGATIVE, BINARY and INTEGER.

To specify the relationships between the PARAMETERS and VARIABLES, the user must declare and define the relationships using the EQUATION keyword. To distinguish between the assignment operator '=' and the equality type of relationship, the latter one is specified with the following symbol: `=E=`.

Since the user has to explicitly declares the types of all variables of the model, it is easier for *GAMS* to handle intermediate expressions and carry out type checking. On the other hand, the user has to type in more information and increases the chances of getting a "compilation" error.

User Control

Once the model has been formulated, the solution process can be handled with very little user intervention. When invoking the solver, the user has to specify the name of the model and the problem class of the model — NLP (Non-Linear Programming) or LP (Linear Programming) in the PC version and LP, NLP, DNLP (NLP with some discontinuous derivatives), MP (Mixed integer Programming) or RMIP (Relaxed MIP) — the MAXIMIZING or MINIMIZING keyword, and the variable to be maximized or minimized. *GAMS* checks to see if the model corresponds or not to the problem class specified by the user before invoking the appropriate solver.

The user may specify which solver to use by using the "`OPTION`" keyword. For example, the user may specify which solver to use, instead of letting *GAMS* choose it. The user can also specify other options like the maximum CPU time allowed (`RESLIM`), the maximum number of line searches (`ITERLIM`) and some solver-specific parameters like the damping parameter (`OPTION REAL1`) and the penalty parameter (`OPTION REAL2`) in MINOS 5.0.

Besides these options, *GAMS* tends to be a rather closed system allowing relatively little user control.

3.2 GINO

GINO — for Generalized INteractive Optimizer — has been under development by Cunningham and Schrage [18].

GINO has a PC version available, which is very similar to *LINDO*, except that it can handle nonlinear programming problems.

Mathematical Programming Solvers Used

The solver used by *GINO* is *GRG2*, which uses a version of the Generalized Reduced Gradient (*GRG*) algorithm. The *GRG* algorithm was first developed by Jean Abadie, in the late 1960's and is also one of the solvers used by *IFPS/OPTIMUM* and the mainframe version of *GAMS*.

GINO does not support the concept of an open library of solvers.

System Architecture

Unfortunately, very little is known about *GINO*'s internal architecture except that it must have some kind of interface with its solver and it seems that this interface is internal. That is, it does not communicate with the solver through external ASCII files.

Modeling Language Features

GINO's modeling language is very simple — like *LINDO* — and therefore very easy to learn. However, *GINO* does not support indices, which means that all the constraints have to be entered in scalarized form. There is no shorthand notation for handling arrays. These means that for large problems, the generation of the input is very hard and tedious. It also means that *GINO* is not well suited for separating model structure from detailed data.

A model in *GINO* is a set of equations, each one separated from the next by a semi-colon. The objective function is indicated by preceding it with the "`MIN =`" or "`MAX =`" keywords. The model is ended by an "`END`" keyword. *GINO* can also be used to solve systems of equations and evaluate expressions. This is done simply by not specifying any objective function. If there is more than one solution to the set of equations *GINO* will find only one of them.

Model formulations in *GINO* tend to be rather cryptic since variable names tend to be short and comments are not considered an intrinsic part of the model. This means that if comments — which are introduced by typing an exclamation sign — are entered in interactive mode, they will disappear when the model is saved. Only if the model is created using an external text editor, will the comments be preserved.

On the other hand, *GINO*'s simple language can be used to express almost any kind of model and it is very good for relatively small problems.

Model Management Features

GINO is very poor in Model Management features. It really seems intended for "one-shot" problems.

Models in *GINO* are very problem-specific, and if the problem changes significantly, it is probably necessary to generate the whole input file again.

Data Management Features

Data Management in *GINO* is also very primitive. The data are **NOT** separated from the general model structure.

User Interface

The user interface of *GINO* is very similar to *LINDO*'s user interface. It is not truly an interactive interface because it is not very supportive during the process of creating the model. In fact, it seems to be better to formulate the model using an external text editor and then import it into *GINO*.

GINO has on-line help available and is command driven. However, *GINO*'s PC version does allow the use of "pointing" to choose a file to be imported.

Intermediate Expression Handling

GINO does not support the use of intermediate expressions. One can always use "fixed" variables, but the performance probably would deteriorate.

User Control

GINO requires very little user control in order to solve a problem. However, as with most of nonlinear programming solvers, the user may have to guess an initial solution if the solver runs into trouble. Also, *GINO*'s solver — GRG2 — can only find local optima, and it is the user's responsibility, by specifying different starting solutions, to pursue a global optimum solution.

For the more technical users, *GINO* has the SETP command which allows the user to override the default tolerances used by the GRG2 optimizer. In order to use this command, the user has to be familiar with the GRG2 optimizer.

3.3 GXMP

The Generalized eXperimental Mathematical Programming system (*GXMP*) was developed by Dolk as a Model Management system [7]. Although *GXMP* was not intended to be a fully general Model Management system, it was intended to show that abstractions can be implemented feasibly in a useful, workable Model Management system.

Mathematical Programming Solvers Used

The *GXMP* system uses the *XMP* library of FORTRAN subroutines for solving linear programming problems. However, there is no restriction for using other linear programming algorithms, as long as they accept input in sparse matrix form.

GXMP does not solve nonlinear programming problems.

System Architecture

Unfortunately, there was not much detailed information available on the system design of the *GXMP* system except that the programming language used was FORTRAN, and that the DBMS module — ADBMS — was also written in FORTRAN.

The *GXMP* system consists of the following components:

1. CODASYL data bases:
   (a) Dictionary/Directory data base: the dictionary catalogs what entities are in the system (specifically, model, parameter, procedure, abstraction, and data base instances) and the directory catalogs where these entities are located;
   (b) Abstraction data base: stores LP model abstractions;
   (c) Procedure data base: stores *XMP* subroutines;
   (d) Equation data base: stores model equations expressed in modeling language;
   (e) Parameter data base: stores data for parameters in equations;
2. Modeling Language: mathematics-like language for expressing constraints and objective functions;
3. Model Translator: transforms equations in modeling language into *XMP* matrix for simplex solution;
4. Model Solver: uses model abstraction to generate job stream of *XMP* subroutine calls;

5. Solution Reporter: converts *XMP* solution matrix into report associating user-specified parameter names with solution names;

6. Menu Dialog: performs editing functions (creation, update and display) for equations, procedures, and abstractions.

Modeling Language Features

The *GXMP* modeling language was designed to satisfy the following assumptions [7]:

1. Algebraic equations are a more concise and "natural" way of representing linear programming models than matrices;

2. Model and data independence must be enforced;

3. The language must be usable for, and easily extensible to, applications other than linear programming.

The *GXMP* modeling language bears a strong resemblance to the *XML* language suggested by Fourer [10].

In general, *GXMP* attempts to follow, as much as possible, the algebraic notation. However, user-supplied variable names may be used instead of generic mathematical variables.

There are some important restrictions in the *GXMP* language, especially regarding index sets. *GXMP* does not provide full capabilities for specifying subsets of index values. Particular values of an index, other than the first and the last, cannot be specified in an equation. For example, indices cannot be functions of other functions which would be particularly valuable in certain network problems. Also, equations which have two or more summations over the same index but over different values of that index cannot be expressed in the *GXMP* modeling language.

Model Management Features

The model management capabilities of *GXMP* are one of its strongest points, which is not surprising since *GXMP* was designed as a Modeling Management system.

As mentioned earlier, *GXMP* has 5 data bases. The Equation and Parameter data bases contain the model description, the Abstraction and Procedure data bases contain the model manipulation information and the Dictionary/Directory data base contains the model control information.

The model description in the *GXMP* system is stored in two separate data bases: the equations data base and the parameter data base. Each equation data base in *GXMP* contains a model instance represented as equations expressed in the *GXMP* modeling language. The data values of the model are kept in parameter data bases which are structured so that functional relationships between parameters are preserved. In particular, parameters and decision variables which depend on index-sets are logically linked to those index-sets within the data base.

The alteration of one or more data values in a parameter data base has no impact on any equation data base. Conversely, modifying equations in an equation data base does not affect any existing parameter data base.

The model manipulations are stored in the Abstraction data base and the Procedure data base. The abstraction data base consists of objects, procedures, and assertions (or rules) which are all described in first order predicate calculus[3]. Although abstraction data bases provide a knowledge base capability, there are currently no inference mechanisms in *GXMP* which work from this knowledge. The main purpose of the abstraction data base is to provide a SOLVE procedure predicate that indicates which matrix generation and *XMP* routines are needed to solve model instances of the abstractions. The *XMP* routines on the other hand, are stored in the Procedure data base.

Finally, the Dictionary/Directory (D/D) data base provides management and administrative information about models and data. The D/D can help answer questions of the form:

1. "If we add 2 more WAREHOUSEs, which models will this affect?"
2. "What parameters does the PRODUCTION_INVENTORY model use?"
3. "List the current inventory of LP models."
4. "Who is responsible for maintaining the PRODUCT_MIX model?"

[3]For a more complete description of the model abstraction concept see [7]

5. "Which models is Bill Jones responsible for?"

The D/D is essentially a meta-data base in that it contains descriptive information about the model and data entities in the system as opposed to actual data values about those entities.

Data Management Features

The data management features in *GXMP* are very good since data are kept in separate data bases and the system includes a built-in *DBMS*. This *DBMS* has a Dictionary/Directory data base which provides management and administrative information about models and data.

The *DBMS* used by *GXMP* is *ADBMS*, which is written in FORTRAN and conforms to a subset of the CODASYL data base standards.

The Dictionary/Directory (D/D) in *GXMP* is active. That is, all transactions initiated by the user are routed to the D/D first to determine which entities must be accessed, whether they exist, and if so, where they are located. The D/D catalogs all entities in the system except the equations. Equations are not cataloged because they are model-specific.

User Interface

The *GXMP* system is intended for three classes of users: the modeler, the decision maker and the model administrator. The modeler's interface is mainly the modeling language and it is assumed that this user is more technically oriented. The decision maker is the person for whom the model is being designed. The model administrator is seen as the model counterpart to the data base administrator in a data management environment. He creates and maintains the model abstractions that drive the system. Since it is assumed that the decision maker will probably have very little previous computer experience, a menu dialog is used in *GXMP* to promote the overall "user friendliness" of the system.

The menu dialog in *GXMP* consists of menus arranged in a hierarchy of roughly three levels. The dialog moves up or down the hierarchy depending on the success or failure of the current operation specified. The three levels may be characterized as activity, instance and operation. The beginning level asks the user to specify the type of activity to be engaged in (Equation Operations, Parameter Operations and Solve Model). Once the user specifies the activity, the next level queries for a particular instance of the entity involved (e.g., a model name if the user specified the Solve Model activity). Once the instance has been specified and accessed, the third level offer various operations that can be performed on the instance.

Intermediate Expression Handling

GXMP does not allow intermediate expressions. Every equation must contain at least one decision variable and constitute either a constraint or the objective function. This was done to expedite the transformation process.

User Control

The user can achieve a good degree of control by using the Model Abstractions. In fact he *must* create a model abstraction, which can be a fairly difficult task. The assumption behind this decision is that the model will be created by a technically oriented user, who plays a role somewhat similar to a knowledge engineer in the development of an expert system. This person will set up the equations of the model and the model abstraction. Then the decision maker can use the system without having to modify the equations nor the abstraction, unless he knows how to do that.

3.4 HEQS

HEQS — Hierarchical EQuation Solver — is a set of tools for numerically solving sets of algebraic equations from their description in a text file [6]. It allows users to define a model as a set of equations that can be analyzed and solved with minimal user intervention. *HEQS* can deal with unsubscripted and or multiply subscripted (array) variables. *HEQS* commands can either be used interactively, to define and solve models, or as a set of *UNIX* operating system high-level algebraic tools for building applications that require model solving.

Mathematical Programming Solvers Used

HEQS was designed to solve sets of nonlinear simultaneous equations. It is the only system surveyed that does not include an optimization solver[4]. However, *HEQS* can solve very large sets of nonlinear equations by decomposing them (using strong component algorithms) into the smallest possible sets of simultaneous equations, each more easily inspected and solved. *HEQS* solves the nonlinear equations by iteration, and requires that users provide an estimated initial value for any variable whose value is found in this way.

HEQS also provides a Goal-Seeking capability which allows the determination of the data values in the model that guarantee particular output values for the solution.

System Architecture

In [6], the architecture of *HEQS* is very well described. "The basic architecture of the system was chosen to model corresponding steps in the equation-solving process." The front end is a macro processor or scalarizer module that parses and then *scalarizes*[5] compactly written scalar or tensor equations into their scalar text equivalents; it then abstracts and stores as a graph the dependent (left-hand-side) variables and its independent (right-hand-side) variables for each scalar equation. These scalar equations and their graph representation are passed to the hierarchy finder module that decomposes them into the smallest possible sets of simultaneous equations, and determines an order for their solution. Finally, the solver module finds a numerical solution to the model after reading in the scalarized equations and the order in which to solve them.

Among the advantages cited in [6] for using a modular architecture, probably the most important one is the following:

> "It allows the crucial separation of the scalarizer, which defines the language for writing equations, from the solver, which finds the

[4]Of course a user could use *HEQS* to solve for the first order conditions. However, many users may not know how to derive the Kuhn-Tucker conditions or may find it inconvenient. Also, if the user is solving a linear programming problem, the computation time of using this approach will be much greater than for using a linear programming code.

[5]"Scalarise" in this context means converting a symbolic representation of a family of equations into a scalar representation of the same equations. That is, this module converts implicitly defined sets of equations into a number of explicit equations.

solution. This makes possible the independent enhancement or replacement of either, and has already proved extremely useful."

Another advantage cited in [6], is that each module can be independently debugged thanks to the weak coupling of all the modules, which communicate only by intermediate files. This is a big advantage when developing large and complex software systems.

In *HEQS*, all commands are implemented as separate programs. The most important of these are the following:

`cheq` (CHeck EQuations) is the primary gateway into *HEQS*' modeling environment. It reads the model from a standard input file, parses it, reports syntax errors, scalarizes as necessary and builds a dependency graph — stored in an intermediate file called OUTGRAPHFILE— for future model analysis.

`seq` (Sequence EQuations) reads `cheq`'s output — OUTGRAPHFILE — and decomposes the model into linked simultaneous subsets of equations and then determines the order for solution that allows them to be solved subset by subset. The order is stored in another intermediate file called ORDERFILE.

`canislv` (CAN I SoLVe) uses the order for solution in ORDERFILE to test whether the model is underdetermined.

`slv` (SoLVe) reads the dependency graph produced by `cheq` and the order produced by `seq` to determine a numerical solution to the model. This solution is stored in an intermediate file called SOLGRAPHFILE.

Other commands are: `wgl` (variable impact analysis), `repslv` (repetitive solution), `goalsk` (goal-seeking), `sens` (sensitivity analysis), `whatif` (what if analysis), `addfunc` (add a user function) and `compmodel` (compile a model). Most of these commands — like `repslv`, `goalsk`, `sens`, `wgl` and `whatif` — require additional input from the user. The `addfunc` and `compmodel` are considered to be "meta" commands since they modify *HEQS* `slv` command.

Modeling Language Features

HEQS' modeling language is nonprocedural. The equations can be written in any order which can help make the model easier to understand. It also has many of the features and some of the syntax of the C language. For example, *HEQS'* modeling language supports the "`DEFINE`" keyword which is used to define macros at the beginning of the input file. Also, comments can be included as in the C language —using `/*` and `*/`— and boolean expressions follow the C language syntax.

A model in *HEQS* is a set of equations, in which a variable appears on the left hand side of the equal sign and an expression appears at the right hand side. Variables can be subscripted by appending a list of indices enclosed by square brackets to the variable name. The list of indices are separated by commas. In order to reference a subscripted variable, a list of index values — separated by blanks — is specified in the square brackets. The index values can be any identifier. However, if integer numbers are used, 1:n can be used as a shorthand notation for 1 2 ... n.

In *HEQS'* modeling language, it is not necessary to declare the size of the arrays, or the index values that will be used. However, by using the `DEFINE` command mentioned before, it is possible to associate a set of index values to a name so that the name can be used as a symbolic index. In fact, multiple index value sets can be assigned to the same index. For example, MONTH can be defined as 1:12, while SUMMER can be defined as 6 7 8, WINTER as 12 1 2 and so on. If the left hand side variable is a subscripted variable with multiple index values, this effectively defines a family of equations which is very useful to deal with the block structure of the model.

The expression on the right hand side of the equation can have an `IF (boolean_expression) THEN expression1 ELSE expression2` form to handle conditional equations. This is very good for handling the exceptions to the block structure of the model.

A very useful built-in function in *HEQS* is the `SUMOF` function which adds elements of subscripted variables. `SUMOF` adds all the elements of the array indicated by the index values appearing in the index list.

A useful operator in *HEQS'* modeling language is the left shift operator: "<<". This operator can be used to shift time series to the left by some number of units. For example, if MONTH is defined as 1 2... 12, MONTH<<1 results in 0 1 ... 11.

Model Management Features

HEQS does not provide very good support in this area. However, a very interesting feature of *HEQS* is the ability of "compiling" a model. Once the model has been developed, it can be translated into a compilable C program that solves the particular model more rapidly than the standard *HEQS* `slv` command which interpretively solves a general model.

Data Management Features

The data management features in *HEQS* are also very poor. The data for the model have to be included in the input file. This makes it difficult to separate the data from the general model structure. However, a certain degree of separation can be achieved by using the `DEFINE` capability of *HEQS*.

User Interface

HEQS is actually a set of tools, whose main input — as stated in [6] — is: "... a model — an easily edited text description of a set of equations (and associated comments) to be solved. Models are most naturally kept in *UNIX* system files."

HEQS programs are tailored to the *UNIX* system and its shell.[6] The programs can be used in two modes:

1. As simple high-level commands to solve models, or

2. Combined with the control structures of the shell, to provide a model-solving language for use by applications system builders that require equation solving capabilities for their users.

In other words, *HEQS*' user interface is very simple and plain: It constitutes a very good example of what has been called "little languages" [1]. However, coupled with a *UNIX* system shell, it can be used to easily tailor a good interface for an specific class of users.

[6]The commands are all implemented as separate programs that communicate with each other through intermediate files.

Intermediate Expression Handling

Since *HEQS* solves sets of equations, intermediate expressions are just trivial equations which are naturally detected in the dependency and ordering phase of the solution procedure. Obviously, intermediate expression equations are 'solved' first.

User Control

Since *HEQS* is a set of commands which can be put together in a *UNIX* script, the degree of control by the user is very flexible. However, since the system is primarily oriented toward non-programmers, it does not seem to allow user control at a very technical level.

It is important to note that *HEQS* does require an initial value from the user for variables which are determined by nonlinear equations.

3.5 IFPS/OPTIMUM

IFPS (Interactive Financial Planning System) is the best selling mainframe financial planning language. *OPTIMUM* is a version of *IFPS* that incorporates linear, nonlinear and integer programming capability [23].

The *IFPS* Planning Language is probably the most popular planning language in the market. It has been in use for many years on mainframe computers and there also has been a PC version — *IFPS/PERSONAL* — available for some time. There are some differences among the different versions that exist, which makes it difficult to talk in general about *IFPS*. In particular, the mainframe version of *IFPS* has many differences from the PC version. Even though *IFPS/PERSONAL* does not yet include optimization capabilities, it does have many interesting features not found in mainframe *IFPS* and it will probably have optimization capabilities in the near future. Therefore, many features of *IFPS/PERSONAL* will be included in the survey, even though they may not be present in the mainframe version of *IFPS*.

IFPS uses a matrix format (similar to a spreadsheet), but it does not allow the user to visualize and manipulate the matrix in a direct way as spreadsheet programs do. Also, *IFPS* (as well as spreadsheets) only provide evaluation capabilities. However, a few years ago, the *OPTIMUM* interface was added allowing mainframe *IFPS* users to access linear, nonlinear and integer programming solvers.

Mathematical Programming Solvers Used

In *IFPS/OPTIMUM*, linear programs are solved by a sparsity oriented primal simplex code, and nonlinear programs by the *GRG2* optimization system.

The *IFPS/OPTIMUM* system uses an "analytic" approach for computing partial derivatives. This approach seems to be better than the finite difference approach when solving nonlinear problems with the *GRG2* program. Partial derivative derivatives are used to detect if the model is linear or nonlinear. If the model is linear, then the partial derivatives can be used as the coefficients for the solver, and if the problem is nonlinear, the partial derivative have to be re-evaluated each time *GRG2* requires it. This rather close integration probably makes it difficult to support the concept of an open library of solvers (*Feature (b)* in section 1.2 of [13]).

System Architecture

In [23] the *IFPS/OPTIMUM* system is described as being "...a modular system composed of over 60 FORTRAN subroutines which perform the following major steps:

1. Read the user's specification of which model variables are decisions, constraints, and the objective;

2. Classify the remaining model variables endogenous or exogenous in both regular and special columns;

3. Compute the nonzero elements of the endogenous and control Jacobian matrices $\partial G/\partial y$ and $\partial G/\partial u$, flagging elements which depend on y or u;

4. Determine if the optimization problem is linear or nonlinear;

5. Compute the reduced gradient $\partial F/\partial u$ for each objective or constraint function F (once for linear problems, when needed for nonlinear problems);

6. Transmit the reduced gradients and other problem data to the appropriate optimizer and, in case of a nonlinear model, communicate with it during the solution process;

7. Retrieve and report the results of the optimization."

The fact that during the solution of a nonlinear problem, the system must compute the needed partial derivatives to the solver, suggests that there must be a close relationship between the solver and the rest of the system. If not, the solution time for nonlinear problems would be too high.

Modeling Language Features

IFPS/OPTIMUM's modeling language is basically the *IFPS* planning language. This language is very powerful for handling block structures. However, exceptions to the block structure are more difficult to handle.

The *IFPS* planning language is nonprocedural. That is, the order of the statements does not matter, and there is no need to declare before use. The *IFPS* language is based on a matrix format, where columns often represent time periods but can also correspond to locations, products, etc. Each row of the matrix is used to specify a variable, and for each cell of this matrix, the planning language permits the specification of a rule for computing the value of the variable in that cell. This is very similar to the conceptual framework used by spreadsheet software like *Lotus 1-2-3*. However, *IFPS* has a stronger bias towards arrays of identical horizontal dimension.

Each line in an *IFPS* model is supposed to be a separate statement, unless a continuation marker is present at the end of the line. A particular cell in the matrix can be referred to by typing `MATRIX(var-ref, col-ref)` or `var-ref[col-ref]` in *IFPS/PERSONAL*. Also, it is possible to refer to a set of values corresponding to a particular variable. But due to the lack of indexing capabilities, it is not possible to refer to subsets of values of a given variable.

Model Management Features

IFPS/OPTIMUM does not have very good Model Management features. Due to the lack of indexing capabilities, it is difficult to establish a clear distinction between the general model structure and its data, and therefore *IFPS* models tend to be problem-specific. Usually it is hard to adapt an *IFPS* model to a similar problem but with different data.

However, since the language can use English-like names and is nonprocedural, the models tend to be self-documenting. Also, comments can be inserted. Names in *IFPS* can have spaces in them and can be arbitrarily long even though 'only' the first 63 characters are checked for uniqueness.

Data Management Features

IFPS/OPTIMUM was not really designed to have data management features. However it does have some. There is an `INCLUDE` command which allows adding external model and data files into the model. There is also the "`DATA`" keyword which specifies that certain variables will get their value from an external data file. This allows the creation of a "template" model which can be used with different sets of data. This means that *IFPS* supports to a certain degree the separation between model structure and detailed data. However, it is important to note that this separation is limited because of the lack of indexing capabilities.

User Interface

IFPS' user interface tends to be rather plain, since it was initially developed for mainframe computers. This means that the user specifies the model either by typing it in at the terminal, under *IFPS* control, or by constructing an input file using a text editor and then submitting this file to *IFPS*. While under *IFPS* control, the user may issue different commands to manipulate the model.

The *OPTIMUM* interface is activated by simply typing the "`SOLVE`" command. Previously, the user must have had to specify the objective function —by preceding it with the `MAXIMIZE` or `MINIMIZE` keywords— the decision variable and the constraints. Declaring a variable as a decision variable overrides any definition for that variable given in the model.

Intermediate Expression Handling

IFPS/OPTIMUM is one of the systems that handles intermediate expressions. This may be because it fully recognizes the importance of intermediate expressions and also because *IFPS* can be seen simply as an intermediate expression evaluator.

IFPS/OPTIMUM classifies all the variables into three categories: decision, exogenous and endogenous. Furthermore, endogenous variables are classified as intermediate or problem functions. Finally, intermediate endogenous variables are classified as either extraneous or not extraneous. The exogenous variables are those whose values can be determined, directly or indirectly, by given data. That is, an exogenous variable is defined by other exogenous variables or by data. Therefore, exogenous variables correspond exactly to "intermediate expressions" as they were defined in 2.7.

In *IFPS/OPTIMUM*, all the exogenous variables are identified and their value is computed only once. According to [23], between 35% to 70% of the variables of seven *IFPS* models examined were exogenous. The reduction in the number of variables handled by the solver is crucial when solving nonlinear problems.

User Control

The solution process requires almost no user intervention. The system automatically checks the problem and determines which solver to user: linear or nonlinear. It also builds the appropriate input for the solver, invokes the solver and interprets the solution.

IFPS has a PROFILE system which can be used achieve certain degree of user control. It is not known however, if this system can be used to change certain settings in the *OPTIMUM* system.

4 Conclusion

None of the systems reviewed has all the desirable features proposed by Geoffrion in [12]. In particular, none of the systems is useful for most phases of the entire life-cycle associated with model-based work (*feature (d)*).

Each of the modeling systems surveyed had strengths and weaknesses. There was no clearly "best" modeling system. Each one could be the best choice under different circumstances. For example, *GAMS* is the one that has the most powerful and flexible modeling language. However, *IFPS/OPTIMUM* models are easier to understand for people who are not mathematically knowledgeable. Also, *IFPS/OPTIMUM* has a big *IFPS* user base which means it has many potential users who already know that system's modeling language. *GINO*, on the other hand, has the easiest to learn modeling language, even though it is inefficient for large models.

The fact that none of the systems surveyed has succeeded in creating a modeling environment that supports the entire modeling cycle does not mean these systems have failed, but rather that completely satisfying these features may be very difficult. This degree of difficulty suggests using a modular approach to develop the system.

There is no doubt that the existing modeling systems will continue to improve, and that new and better ones will appear. In fact, AT& T is developing a system, called *AMPL* (**A** **M**athematical **P**rogramming **L**anguage), that appears to be very powerful.

A direction that has not yet been successfully exploited, but that appears to be quite promising, is the application of Artificial Intelligence concepts in the development of modeling systems. In particular, it would be very useful to have an expert system that would be able to analyze a model representation and its data to determine to which class of models this particular model belongs to, which solver or solvers will be the best for the specific model and also what are the appropriate settings for the solver's parameters. It could also determine if it is necessary to scale the data, and determine the quality of the solution obtained. All of these things are currently carried out by an *expert* problem solver and it may be possible to extract the knowledge used by these experts to build an expert system into the modeling system.

References

[1] Bentley, J. (1986). "Programming Pearls: Little Languages," *Communications of the ACM*, **29**:8 (August), pp. 711–721.

[2] Bisschop, J. and A. Meeraus (1982). "On the Development of a General Algebraic Modeling System in a Strategic Planning Environment," Mathematical Programming Study 20 (October), North-Holland, Amsterdam, pp. 1–29.

[3] Brooke, A., A. Drud and A. Meeraus (1985). "High Level Modeling Systems and Nonlinear Programming". *Nonlinear Optimization, 1984*, pp. 178–198, SIAM, Philadelphia.

[4] Burger, W.F. (1982). "MLD: A Language and Data Base for Modeling," IBM Research Division, San Jose, Research Report RC 9639 (#42311), September 14.

[5] Cardenas, A. (1984). *Data Base Management Systems*, Allyn and Bacon, Inc., Boston, Mass.

[6] Derman, E. and E.G. Sheppard (1985). "HEQS — A Hierarchical Equation Solver," *AT&T Technical Journal*, **64**:9 (November), pp. 2061–2096.

[7] Dolk, D.R. (1986). "A Generalized Model Management System for Mathematical Programming," to appear in *ACM Transactions on Mathematical Software*.

[8] Drud, A. (1983). "Interfacing Modeling Systems and Solution Algorithms," *Journal of Economic Dynamics and Control*, 5, pp. 131–149.

[9] Ellison, E.F.D. and G. Mitra (1982). "UIMP: User Interface for Mathematical Programming," *ACM Transactions on Mathematical Software*, 8:3 (September), pp. 229–255.

[10] Fourer, R. (1983). "Modeling Languages Versus Matrix Generators for Linear Programming," *ACM Transactions on Mathematical Software*, 9:2 (June), pp. 143–183.

[11] Fourer, R., Gay, D. M. and B. W. Kernigham (1987). "AMPL: A Mathematical Programming Language," Computing Science Technical Report No. 133, January, AT& T Bell Laboratories, Murray Hill, New Jersey 07974.

[12] Geoffrion, A. (1987). "Introduction to Structured Modeling," forthcoming in *Management Science*, May.

[13] Geoffrion, A. (1987). "Modeling Approaches and Systems Related to Structured Modeling," Working Paper No. 339, Graduate School of Management, UCLA, February.

[14] Geoffrion, A. (1986). "Integrated Modeling Systems," Working Paper No. 343, Graduate School of Management, UCLA, November.

[15] Gill, P., W. Murray, M. Saunders and M. Wright. (1984). "Trends in Nonlinear Programming Software," *European Journal of Operational Research* 17, pp. 141–149.

[16] Katz, S., L.J. Risman and M. Rodeh (1980). "A System for Constructing Linear Programming Models," *IBM Systems Journal*, 19:4.

[17] Kendrick, D.A. and A. Meeraus (1985). *GAMS: An Introduction*, Draft Book, The World Bank, February.

[18] Liebman, Schrage, Lasdon and Waren (1984). *Applications of Modeling and Optimization with GINO*, Draft Book, Graduate School of Business, University of Chicago, November.

[19] Liffick, B. W. (1985). *The Software Developer's Sourcebook: from Concept to Completion*, Addison-Wesley, Reading, Mass.

[20] Lucas, C. and G. Mitra (1985). *CAMPS: Preliminary User Manual*, Department of Mathematics and Statistics, Brunel University, Middlesex, U.K., July.

[21] Palmer, K. (1984). *A Model Management Framework for Mathematical Programming*, Wiley, New York.

[22] Rosenthal, R. (1986). "Personal Computing & OR/MS: Review of the GAMS/MINOS Modeling Language and Optimization Program," *OR/MS Today*, **13**:3, pp. 24–32.

[23] Roy, A., L. Lasdon and J. Lordeman (1986). "Extending Planning Languages to Include Optimization Capabilities," *Management Science*, **32**:3 (March), pp. 360–373.

[24] Schittkowski, K. (1985). "EMP: A Software System Supporting the Numerical Solution of Mathematical Programming Problems," Working Paper, Institut fur Informatik, Universitat Stuttgart.

[25] Schrage, L. (1984). *Linear, Integer and Quadratic Programming with LINDO,* The Scientific Press, Palo Alto, California.

[26] Van Der Eijk, P. and H. Patton (1983). *Description of GAMS Machine*, Draft.